Object orientation

• identity
• abstraction
• classification
• encapsulation
• inheritance
• polymorphism
• persistence

Objects and classes

• Every object has a name (also called a reference or handle).
• Objects can have attributes (such as color, size, location).
• Objects can have operations or behaviors (such as takeoff, land, repair).
• Each object is an instance of a class.
• A specific implementation of an operation for a certain class is called a method.
Elephant

- Color: text
- Number of tusks: Integer
- Location: text
- Weight: float
- Height: float
- move_to (location)
- wash (date)
- feed (amount, date, time)
Table 6.1: Tendency for change when using OO paradigm (Jacobson et al. 1995)

<table>
<thead>
<tr>
<th>Characteristic of software product/project</th>
<th>Probability for change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objects derived from the application</td>
<td>low</td>
</tr>
<tr>
<td>Long-lived information structures</td>
<td>low</td>
</tr>
<tr>
<td>Passive object’s attribute</td>
<td>medium</td>
</tr>
<tr>
<td>Sequences of behavior</td>
<td>medium</td>
</tr>
<tr>
<td>Interface with the outside world</td>
<td>high</td>
</tr>
<tr>
<td>Functionality</td>
<td>high</td>
</tr>
</tbody>
</table>
OO design

- Usually uses an OO requirements representation
- System design identifies and represents objects and classes, plus details of each object's attributes and behaviors.
- System design also identifies interactions and relationships.
- Program design inserts computational features in the models.
- Program design inserts class library details.
- Program design considers nonfunctional requirements to enhance design.
Use cases

• Diagrams have four elements:
  – actors
  – cases
  – extensions
  – uses
Identifying participants

• What users or groups use the system to perform a task?
• What users or groups are needed so that the system can perform its functions?
• What external systems use the system to perform a task?
• What external systems, users or groups send information to the system?
• What external systems, users or groups receive information from the system?

Royal Service Station

1. The Royal Service Station provides three types of services to its customers: refueling, vehicle maintenance, and parking. That is, a customer can add fuel to the tank of his or her vehicle (car, motorcycle, or truck), can have the vehicle repaired, or can park the vehicle in the station parking lot. A customer has the option to be billed automatically at the time of purchase (of fuel, maintenance, or parking) or be sent a monthly paper bill. In either case, customers can pay using cash, credit card, or personal check. Royal Service Station fuel is sold according to price per gallon, depending on whether the fuel is diesel, regular, or premium. Service is priced according to the cost of parts and labor. Parking is sold according to daily, weekly, and monthly rates. The prices for fuel and maintenance services, parts and parking may vary; only Manny the station manager can enter or change prices. At his discretion, Manny may designate a discount on purchases for a particular customer; this discount may vary from one customer to the another. A 5% local sales tax applies to all purchases.
2. The system must track bills on a month-to-month basis and the products and services provided by the gas station on a day-to-day basis. The results of this tracking can be reported to the station manager upon request.

3. The station manager uses the system to control inventory. The system will warn of low inventory and automatically order new parts and fuel.

4. The system will track credit history and send warning letters to customers whose payments are overdue. Bills are sent to customers on the first day of the month after the purchases are made. Payment is due on the first day of the succeeding month. Any bills not paid within 90 days of billing date will result in cancellation of the customer’s credit.

5. The system applies only to regular repeat customers. A regular repeat customer means a customer identified by name, address, and birth date who uses the station’s services at least once per month for at least six months.

6. The system must handle the data requirements for interfacing with other systems. A credit card system is used to process credit card transactions for products and services. The credit card system uses the card number, name, expiration date, and amount of the purchase. After receiving this information, the credit card system confirms that the transaction is approved or denied. The parts ordering system receives the part code and number of parts needed. It returns the date of parts delivery. The fuel ordering system requires a fuel order description consisting of fuel type, number of gallons, station name, and station identification code. It returns the date when fuel will be delivered.
7. The system must record tax and related information, including tax paid by each customer, as well as tax per item.

8. The station manager must be able to review tax records on demand.

9. The system will send periodic messages to customers, reminding them when their vehicles are due for maintenance. Normally, maintenance is needed every six months.

10. Customers can rent parking spaces in the station lot on a day-to-day basis. Each customer must request from the system an available parking space. The station manager can view a monthly report summarizing how many parking spaces were available or occupied.

11. The system maintains a repository of account information, accessible by account number and by customer name.

12. The station manager must be able to review accounting information upon demand.

13. The system can report an analysis of prices and discounts to the station manager upon request.

14. The system will automatically notify the owners of dormant accounts. That is, customers who did not make service station purchases over a two-month period will be contacted.

15. The system cannot be unavailable for more than 24 hours.

16. The system must protect customer information from unauthorized access.
UML and the OO process

- Workflow diagrams
- Object model
- Sequence diagrams
- Collaboration diagrams
- Package diagrams
- Component diagrams
- Deployment diagrams
First cut at object classes

- Structures
- External systems
- Devices
- Roles
- Operating procedures
- Places
- Organizations
- Things that are manipulated by the system to be built
Guidelines for building classes

- What needs to be “processed” in some way?
- What items have multiple attributes?
- When do you have more than one object in a class?
- What is based on the requirements themselves, not derived from your understanding of the requirements?
- What attributes and operations are always applicable to a class or object?

Guidelines for identifying behaviors

- Imperative verbs
- Passive verbs
- Actions
- Things or reminded events
- Roles
- Operating procedures
- Services provided by an organization
### Chapter 6

#### Class name: Bill

<table>
<thead>
<tr>
<th>Attribute: type = initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>issue_date : Date</td>
</tr>
<tr>
<td>payment_date : Date</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation(arg list): return type</th>
</tr>
</thead>
<tbody>
<tr>
<td>price()</td>
</tr>
<tr>
<td>taxes()</td>
</tr>
<tr>
<td>customer()</td>
</tr>
<tr>
<td>purchases()</td>
</tr>
<tr>
<td>add_to_bill(customer, amount, date)</td>
</tr>
</tbody>
</table>

---

#### Inheritance (is-a)

**Superclass**

**Supertype**

<table>
<thead>
<tr>
<th>Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discount_rate: number</td>
</tr>
<tr>
<td>price()</td>
</tr>
</tbody>
</table>

**Subclass**

**Subtype**

<table>
<thead>
<tr>
<th>Parking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Price: number = 5.00</td>
</tr>
</tbody>
</table>
Chapter 6

Order

Customer

Ordered item

Salesperson

1 receives

1..*

includes

1..*

has

association

composition

aggregation

dependency

navigation

Chapter 6
Chapter 6

class A

class B

qualified association

class qualifier

Chapter 6
Class name: refuel
Category: service
External documents:
Export control: Public
Cardinality: n
Hierarchy:
  Superclasses: services
  Associations:
    fuel in association <name>
Operation name: price
Public member of: refuel
Documentation:
  // Calculates fuel final price
Preconditions:
  gallons > 0
  Object diagram: (unspecified)
Semantics:
  final_price = gallons * price
  Object diagram: (unspecified)
Concurrency: sequential

Public interface:
  Operations:
    price
Private interface:
  Attributes:
    gallons
    price
Implementation:
  Attributes:
    gallons
    price
State machine: no
Concurrency: sequential
Persistence: transient
Chapter 6

Service Station

Customer

Credit Card System

Purchase

Refuel

refuel()

verify customer (credit_card_num, amount)

pay_by_cash (credit_card_num, amount)

cancel credit card transaction

transaction okay

new_purchase (customer, refuel, date, gallons)

new_refuel (customer, date, gallons)

Chapter 6

Customer

1: parking

3: parking_at (location)

4: new_purchase (customer, parking, date, location)

Parking Space

2: next_available ()

Purchase

5: new_parking (customer, location)

Service Station

Parking
Chapter 6

Part
- part_number
- price
- discount_rate
- min_quantity
  - min_quantity = 2
  - current_quantity
- max_quantity

Fuel
- min_quantity = 100
- current_quantity

[time <= 7]

[authorized]

Authorized

[payment ok]

Purchased

[amount of fuel >= minimum]

[delivery of new fuel]

[amount of fuel < minimum]

[difficulty chosen]

[delivery of new parts]
Chapter 6

Inventory

order_fuel ( )
order_part ( )

Activity A
Activity B
output X

Activity C
Activity D
Program design considerations

- Nonfunctional requirements
- Reused components
- Reusable components
- User interface requirements
- Data structure and management details
**Design Aids**

- **Toolkit**
  - A set of related, reusable classes that provide a well-defined set of functionality.

- **Pattern**
  - A template of abstract architectural elements that can be used as a guide to generate the design.

- **Framework**
  - A reuse of part of a domain-specific design. (More specialized than a pattern.)

**Patterns**

- **Creational Patterns**
  - Abstract the instantiation process. They help make a system independent of how its objects are created, composed, and represented.

- **Structural Patterns**
  - How classes and objects are composed to form larger structures. They use the inheritance to compose interfaces or implementations.

- **Behavioral Patterns**
  - Algorithms and the assignment of responsibilities between objects. These patterns characterize complex flow that’s difficult to follow at run-time.
User Interface Design

• Designing a hierarchy of user commands.
• Refining the sequence of user interactions with the system.
• Designing relevant classes in the hierarchy to implement the user interface design decisions.
• Integrating the user interface classes in the overall system class hierarchy.

Chapter 6
Data Management Design

1. Identify the data, data structures, and relationships among them.
2. Design services to manage the data structures and relationships.
3. Find tools, such as database management systems, to implement some of the data management tasks.
4. Design classes and class hierarchies to oversee the data management functions.
Task Management Design

1. Identify the tasks to be performed, and classify them as event- or time-driven.
2. Determine priorities for the tasks. That is, for each pair of tasks, decide which one has priority if the two are invoke simultaneously.
3. Create a tasks to coordinate all other tasks.
4. Design objects for each task, and define the relationships among them.
Order Parts

**Description:** Purpose is to order new parts automatically when there are not enough parts already in stock.

**Priority:** High. It must be activated when the inventory warns that parts in stock are low.

**Included services:** Verify inventory.

**Managed by/Manages:** Service Station System.

**Communicates by:** Modem with the order_parts system.

Observer Pattern*

**Intent**

Define a one-to-many dependency between objects so that when one object changes state, all its dependencies are notified and updated automatically.

**Motivation**

A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects. You don’t want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.

For example, a graph in a spreadsheet needs to be changes whenever the data table that provides the data changes. There may be multiple graphs of different formats. The spreadsheet and the graphs don’t know about one another, but behave as if they do.

* Gamma, et. al. *Design Patterns*, AW 1995
Observer Pattern (2)

Applicability:

Use the Observer pattern in any of the following situations:

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
- When a change to one object requires changing others, and you don’t know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don’t want these objects tightly coupled.

Chapter 6

Observer Pattern (3)
Observer Pattern (4)

Participants:

- Subject
  - Knows its observers. Any number of observers may observe an a subject.
  - Provides an interfacing for attaching and detaching observer objects.
- Observer
  - Defines an updating interface for objects that should be notified of changes in a subject.

Observer Pattern (5)

Participants (cont.)

- ConcreteSubject
  - Stores state of interest to ConcreteObserver objects
  - Sends a notification of to its observers when its state changes.
- ConcreteObservers
  - Maintains a reference to ConcreteSubject objects
  - Stores state that should stay consistent with the subject’s
  - Implements the Observer updating interface to keep its state consistent with the subject’s.
Observer Pattern (6)

Chapter 6

Observer Pattern
Java Implementation*

```java
public interface Observer {
    /** notify observers a change has taken place*/
    public void sendNotify(String s);
}

public interface Subject {
    public void registerInterest(Observer obs);
}
```

* Cooper, Java Design Patterns. AW, 2000

Chapter 6
public class Watch2Windows extends JFrame
implements ActionListener, ItemListener, Subject {
    private Vector observers;
    public Watch2Windows() {
        /* constructs a frame (window) with three radio buttons and a close button. */
    }

    public void itemStateChanged(ItemEvent e) {
        // responds to radio button clicks
        // if the button is selected
        if (e.getStateChange() == ItemEvent.SELECTED)
            notifyObservers((JRadioButton)e.getSource());
    }
}
registerInterest

public void registerInterest(Observer obs) {
   // adds observer to list
   observers.addElement(obs);
}

notifyObservers

private void notifyObservers(JRadioButton rad) {
   // sends text of selected button to all observers
   String color = rad.getText();
   for (int i=0; i< observers.size(); i++) {
      ((Observer)(observers.elementAt(i))).sendNotify(color);
   }
}
ColorFrame

public class ColorFrame extends JFrame implements Observer {
    private Color color;
    private String color_name="black";
    private Font font;
    private JPanel p = new JPanel(true);
    public ColorFrame(Subject s) {
        /* Construct a panel that has a background color and a string with the name of the color */
        s.registerInterest(this);
    }
}

Chapter 6

ColorFrame.sendNotify

public void sendNotify(String s) {
    color_name = s;
    if (s.equalsIgnoreCase("RED"))
        color = Color.red;
    if (s.equalsIgnoreCase("BLUE"))
        color =Color.blue;
    if (s.equalsIgnoreCase("GREEN"))
        color = Color.green;
    //p.repaint();
    setBackground(color);
}
ListFrame

public class ListFrame extends JFrame
implements Observer {
    private JList list;
    private JPanel p;
    private JScrollPane lsp;
    private JListData listData;

    public ListFrame(Subject s) {
        /* Construct a panel containing a scroll pane. */
        s.registerInterest(this);
    }
}

ListFrame.sendNotify

public void sendNotify(String s) {
    listData.addElement(s);
}

Watch2Windows.main

static public void main(String[] argv) {
    Subject theSubject = new Watch2Windows();
    Observer observer1 = new ColorFrame(theSubject);
    Observer observer2 = new ListFrame(theSubject);
}

Chapter 6